REMARKS

Claims 1 and 3-35 are pending in the application. Because the Advisory Action of October 30, 2008 has informed that the response filed on October 9, 2008 has not been entered, the present amendments to the claims are based on the claim set filed on January 23, 2008.

In the present response, claims 1 and 3-35 have been amended to correct informalities, and claim 35 also has been amended to retain consistency with the recitations of claim 1. Reconsideration and reexamination of the pending claims is respectfully requested in view of the present amendments and remarks.

In the Specification

The Office Action has objected to the specification because some of the references incorporated by reference are not U.S. patents or U.S. patent publications. In order to expedite allowance of the application and without restrictive intent, the incorporation by reference at paragraph [0041] of the specification has been removed.

In the Claims

A. The Rejections under 35 U.S.C. 112, Second Paragraph

Claims 1 and 33 have been rejected under 35 U.S.C. 112, second paragraph because of the term "pseudorandom."

It is respectfully submitted that "pseudorandom" is a term readily recognizable by a person skilled in the art. As evidence, copies are enclosed of Paul E. Black, PSEUDO-RANDOM NUMBER GENERATOR, in Dictionary of Algorithms and Data Structures, National Institute of Standards and Technology; and John Viega, PRACTICAL RANDOM NUMER GENERATION IN SOFTWARE, Virginia Tech.

As additional evidence, also enclosed are pages from six other scientific publications, each identified on the respective page, that use the term "pseudorandom."

Moreover, Feldgajer, US 5,832,466, cited in the Office Action, also uses the term "pseudo-random" without further explanation. See Feldgaier, 8:3.

Based on the foregoing, the rejection under 35 U.S.C. 112, second paragraph is respectfully traversed.

B. The Rejections under 35 U.S.C. 103(a)

Claims 1, 3-9, 11-13, 23-25 and 30-35 have been rejected under 35 U.S.C. 103(a) over Buscema, SCIENTIFIC BACKGROUND OF DYNAMIC ADAPTIVE SYSTEMS ("Buscema I") in view of Feldgajer, US 5,832,466 ("Feldgajer").

Claims 10, 14 and 21-22 have been rejected under 35 U.S.C. 103(a) over Buscema in view of Feldgajer and further in view of Lapointe, US 2003/0004906.

Claims 15-17 have been rejected under 35 U.S.C. 103(a) over Buscema in view of Feldgajer and further in view of Boden, US 5,708,774.

Claims 18-20 have been rejected under 35 U.S.C. 103(a) over Buscema in view of Feldgajer and further in view of Boden and of Burke, A GENETIC ALGORITHM TUTORIAL TOOL FOR NUMERICAL FUNCTION OPTIMISATION.

Claims 26 and 28 have been rejected under 35 U.S.C. 103(a) over Buscema in view of Feldgajer and further in view of Rose, US 2002/0178132.

Claim 27 has been rejected under 35 U.S.C. 103(a) over Buscema in view of Feldgajer and further in view of Breed, US 2003/0002690.

Claim 29 has been rejected under 35 U.S.C. 103(a) over Buscema in view of Feldgajer and further in view of Boden and Lapointe.

The rejections under 35 U.S.C. 103(a) are respectfully traversed at least for the following reasons.

With regard to independent claim 1, Buscema I discloses an artificial neural network (ANN) as an example of a dynamic adaptive system. Buscema I further discloses that, to provide a supervised teaching and testing of ANNs, the records of a database of known cases are subdivided in training and in testing databases, and that after such distribution is performed, the ANN that provides the best output distribution during testing in relation to known outputs is selected as the best network.

Feldgajer teaches a system and a method for dynamic learning control in genetically

enhanced back-propagation neural networks, in which an ANN is designed such that an output response to training is dependent on at least a parameter value of the ANN.

In particular, Feldgajer teaches that, as a first step, an ANN population is generated that includes a plurality of individual ANNs having a known architecture that are divided into groups. Each ANN is provided with a unique parameter value (such as learning rate and momentum), selected to provide genetic diversity among the population. A plurality of input stimuli is then applied to the individuals within the groups, and the "best fit" is determined as the parameter value of the individual that provides a closest output response to the expected output response. New parameter values are subsequently assigned to the groups of individuals based on such "best fit," such that the newly assigned parameter values define a second range of values. Such variations are conducted using a genetic algorithm method. This process is repeated until an individual falls within a predetermined tolerance of the expected output response. See Feldgajer, 4:43-5:51; 6:11-20; 7:50-8:57. This method also is summarized in Feldgajer's FIG. 4, reproduced below:

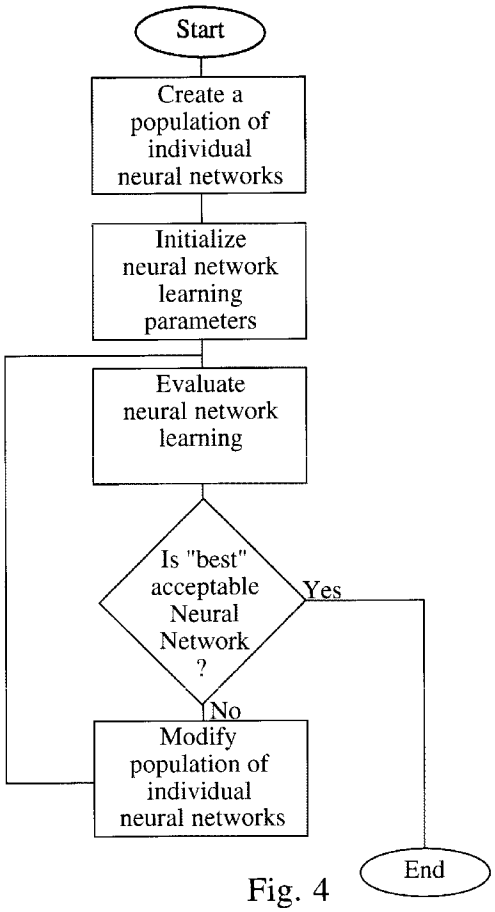

Fig. 4

Neither Buscema I nor Feldgajer, alone or in combination, teach or suggest a method, in

which the starting database is distributed in testing and training subsets and a first generation of prediction algorithms is tested to provide fitness scores for the prediction algorithms of the first generation; in which this first generation of prediction algorithms is subsequently fed to an evolutionary algorithm and a set of one or more prediction algorithms is generated therefrom and a fitness score is assigned to each; and in which new populations of prediction algorithms are generated thereafter, which are trained and tested according to different distributions of the records in the database.

Further, neither Buscema I nor Feldgajer, alone or in combination, teach or suggest that an evolutionary algorithm combines the different models of distribution of the records of the complete data set in one or more training and in a testing sets, each represented by corresponding prediction algorithms trained and tested according to the fitness score calculated in the step discussed in the preceding paragraph.

As shown by the foregoing discussion, Feldgajer teaches combining genetic search techniques with connectionist computation by identifying a “best fit” parameter in an ANN and by exporting such “best fit” parameter to the other ANNs of the population.

Contrary to that, Applicant teaches that that children prediction algorithms are based on a new distribution of records onto the testing and training set, and that such distribution is obtained by merging or mutating the distribution of records of the parent algorithms and not by varying a parameter as in Feldgajer. This causes all children prediction algorithms to be structurally the same but genetically different from as their parent algorithms and from one another, because the children prediction algorithms are generated from different parent algorithms, and, therefore, are trained and tested with training and testing data sets that are different from their parents and from other individuals having different parents.

Claims 3-9, 11-13, 23-25 and 30-35 are believed patentable over Buscema and Feldgajer for the same reasons as claim 1 and for the additional limitations contained therein.

Concerning claims 10, 14 and 21-22; 15-17; 18-20; and 26-29, it is believed that Lapointe, Boden, Burke, Rose and Breed fail to fill the deficiencies of Buscema and Feldgajer, rendering these claims also patentable over the cited references.

Based on the foregoing, the withdrawal of all rejections under 35 U.S.C. 103(a) is respectfully requested.

Conclusion

It is believed that all objections and rejections in the application have been addressed and that the application is now in condition for allowance. A notice to that effect is respectfully requested.

Dated: November 17, 2008

Respectfully submitted,

/Franco A. Serafini/

Franco A. Serafini, Registration No. 52,207
Tel. (858) 456-2898

THEMIS INTELLECTUAL PROPERTY COUNSEL
7660 Fay Ave Ste H535
La Jolla, CA 92037

NIST
National Institute of Standards and Technology

pseudo-random number generator

(algorithm)

Definition: A *deterministic algorithm* to generate a sequence of numbers with little or no discernible pattern in the numbers, except for broad statistical properties.

Also known as PRNG.

Specialization (... is a kind of me.)
linear congruential generator.

See also *randomized algorithm*.

Note: Any computer program is likely to generate pseudo-random numbers, not actually random numbers. This is important when, say, simulations are sensitive to subtle patterns in the "random" numbers used. Hardware-based random number generators are built from parts with naturally random events, such as noise in a diode.

A generator may be "seeded", or initialized, with a random event, such as the current time in milliseconds, to give different sequences every time it is used.

*Do **NOT** use typical "random" number generators for security or cryptographic purposes. Random Numbers from David Wheeler's Secure Programming for Linux and Unix HOWTO, Section 11.3, gives suggestions and guidelines.*

Author: PEB

Implementation

(C++, C, and Fortran). Herbert Glarner's Mersenne Twister MT 19937 (Linoleum). GAMS (C). Using C libraries to get random numbers in a certain range (C) is C FAQ question 13.16.

More information

Random Number Generation and Testing with links to reports, standard tests, and on-going research. ent: a program to test the randomness of bytes in a file. Karl Entacher's thorough review and comparison of A collection of selected pseudorandom number generators with linear structures.

Go to the Dictionary of Algorithms and Data Structures home page.

If you have suggestions, corrections, or comments, please get in touch with Paul E. Black.

Entry modified 21 May 2007.
HTML page formatted Mon May 21 08:45:03 2007.

Cite this as:
Paul E. Black, "pseudo-random number generator", in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 21 May 2007. (accessed TODAY) Available from: http://www.nist.gov/dads/HTML/pseudorandomNumberGen.html

Practical Random Number Generation in Software

John Viega
Virginia Tech
viega@list.org

Abstract

There is a large gap between the theory and practice for random number generation. For example, on most operating systems, using /dev/random to generate a 256-bit AES key is highly likely to produce a key with no more than 160 bits of security. In this paper, we propose solutions to many of the issues that real software-based random number infrastructures have encountered. Particularly, we demonstrate that universal hash functions are a theoretically appealing and efficient mechanism for accumulating entropy, we show how to deal with forking processes without using a two-phase commit, we explore better metrics for estimating entropy and argue that systems should provide both computational security and information theoretic security through separate interfaces.

1. Introduction

Security-critical applications require "random" numbers. Unfortunately, there is no consensus on the best way of obtaining such random numbers. Moreover, there is not a consistent set of requirements or terminology between different solutions.

Ultimately, we would like generators capable of giving "true" random numbers, where all 2^n possible values for an `n` bit datum are equally likely from the point of view of an adversary. I.e., we would like a random number generator to provide information theoretic security, where there is one bit of entropy per bit of generator output.

In practice, that goal often isn't feasible due to the difficulty of harvesting high-entropy data. Instead, it is common to take data that is believed to contain enough entropy for cryptographic security, and use it to "seed" a cryptographic pseudo-random number generator, which is an algorithm that produces a stream of values that one hopes, for all intents and purposes, is indistinguishable from random to any attacker in practice, even if a theoretical attack were possible, given unbounded resources.

Conceptually, producing a pseudo-random stream of numbers from a seed is a well-understood problem. There are simple, efficient cryptographic constructions that leak insignificant amounts of information about the internal state, such as a block cipher run in CTR (counter) mode. Nonetheless, there are numerous practical pitfalls, even when one has a cryptographically sound PRNG. Many of the concerns have been addressed by other work. [15] and [6] both look at the range of attacks that can be launched against random number infrastructures.

Nonetheless, the currently deployed solutions are unsatisfying. The more implementation-oriented systems such as the `/dev/random` and `/dev/urandom` devices on many modern operating systems often fail to facilitate analysis, whereas research-oriented systems tend to ignore some important practical requirements.

Having a secure source of random numbers is a critical assumption of many protocols systems. There have been several high profile failures in random number systems leading to practical problems, such as in the Netscape implementation of SSLv2[10] and a more recent exploit where several online gambling sites had shuffles that were easy to predict[1].

The scope of the problem was shown to be vast when OpenSSL began checking to see whether a seeding function had been called before using its random number generator. The OpenSSL mailing list was flooded with error reports from people with broken software, and similarly with broken suggestions on how to work around the solution. According to [11], many users copy from the OpenSSL test harness, and seed with the fixed value, "string to make the random number generator think it has entropy".

In the embedded space, the problem is even worse, because there is often no access to cheap entropy. Therefore, protocols involving session keys or other data that needs to be random can be at great risk.

Additionally, there are a number of practical issues that should be reconsidered in existing infrastructures. For example, many systems provide only a PRNG, not allowing for information theoretic security. Even among the systems that do try to provide an "interface to entropy", they do not do it in a way that is secure in the information-theoretic sense. For example, Linux's `/dev/random` implementation tries to make sure that

every bit of output contains one bit of entropy, but turns out to have a flaw that makes it unsuitable for producing 192-bit (or higher) AES keys. We discuss this problem is section 6.

In this paper, we discuss a collection of "best practices" for random number generation, many of which are novel. Our recommendations are all implemented in the EGADS random number generation package.

2. Interfaces to Random Numbers

People tend to use pseudo-random numbers instead of numbers that are secure in the information theoretic sense (i.e., truly random numbers with entropy to an attacker) only because pseudo-random numbers are efficient to generate, whereas data with large amounts of entropy tends to not be. One advantage of using truly random numbers is that the difficulty of breaking one random value (by, say, brute-force guessing) is independent of breaking any other value. That is not true with a cryptographic PRNG, where an attacker with enough output can predict future outputs with a probability of 1, assuming enough computational resources. A good PRNG will provide computational security, meaning that an adversary with realistic resources should not be able to distinguish PRNG output from truly random data. However, PRNGs are incapable of providing information theoretic security in the way that a one-time pad can.

We believe that the gap between information theoretic security and computational security argues that systems should not only provide an interface to cryptographically secure pseudo-random numbers, they should also provide an interface to numbers that are believed to be truly random. Even though most applications will not be willing to expend significant resources to get truly random numbers, there are occasions where it can be useful.

For example, there may be some high security data where one would like to ensure that the compromise of one piece of data does not lead to the compromise of other pieces of data, such as long-term, high security keys.

Another consideration is that a system may wish to have more security than a pseudo-random number generator has the strength to provide. For example, the PRNGs provided by most operating systems use cryptography that is based on a 128-bit symmetric cipher or a 160-bit hash function, and thus can never have more than 160 bits of security at any given time. But, one may wish to generate a 256-bit symmetric key, or a public key pair where 256-bit equivalent security is desired.

Note that using two 128-bit PRNGs that are independently seeded does not solve the problem, only doubling an attackers workload, increasing security to 129 bits instead of the desired 256.

Many systems intertwine entropy harvesting and pseudo-random number generation. Those that do not still are generally only concerned with providing a stream of pseudo-random numbers. However, we believe that a good randomness infrastructure should provide at least two interfaces, one that is fast and cryptographically strong, and one where data that is hopefully truly random, with one bit of entropy per bit of output.

There are other types of interfaces that one may wish to provide. Particularly, we see two other useful concepts. First, one may wish to have truly random data if it is available, but still be willing to fall back on pseudo-random data to fill in the rest.

Second, there are times where the state of a pseudo-random number generator need not be secret— it is only important that the outputs be strongly collision resistant. Particularly, the attacker who knows the algorithm for generating outputs should not be able to take a different internal state and be able to produce the same output. This is the case when choosing session IDs for TCP/IP[11], and can be the case when choosing public object identifiers in systems like COM. A cryptographic PRNG will do for these purposes, except when there is not enough entropy available. We prefer to look for other solutions to this problem, as such an API is likely to be misused.

3. Basics

Randomness infrastructures will generally involve a PRNG by necessity, and should always have a component for harvesting entropy (we will refer to data that has the chance of being truly random as entropy because it is a convenient and common term, even if it is a slight misusage in the information theoretic sense).

Even if entropy is only used to seed a PRNG, infrastructures should still harvest their own entropy, because experience shows that pawning the responsibility for entropy harvesting onto clients leads to a large number of systems with inadequately seeded PRNGs.

Entropy gathering should be a separate component from the PRNG. This component is responsible for producing outputs that are believed to be truly random. We will refer to the entropy gather component as the entropy harvester.

Entropy harvesters are responsible for collecting data that may contain entropy, estimating how much entropy has been collected, and turning that data into outputs that are as close to truly random values as

feasible. The last requirement demands some sort of cryptographic post-processing (often called whitening) to try to distribute entropy evenly throughout output values, and to otherwise try to remove any statistical patterns that might have been lingering in the data.

Randomness infrastructures should be designed with a particular threat model in mind. In order to build a threat model, we must first understand the types of attacks that one might try to launch against a system.

At a high level, there are few vectors of attack. First, there may be some way to guess the internal state by observing the outputs (a cryptanalytic attack). Hopefully, such an attack will only apply to a PRNG, but can apply to an entropy harvester if data is mishandled or entropy is overestimated. Second, an attacker may have knowledge of sources that the entropy collector polls. Third, an attacker may be able to introduce malicious data into the entropy collector. Finally, there may be some sort of side channel allowing the attacker to read some or all of the internal state of the generator. This may be a timing channel, but it is more likely to be due to environmental compromise unrelated to the generator. For example, if an attacker can look at kernel memory, she may be able to read the state of the entire randomness infrastructure directly.

Generally, a threat model will assume that the attacker cannot break strong cryptographic algorithms with 80-100 bits of security, meaning that a good, well-seeded PRNG makes the first class of attack uninteresting. For the rest of the possible threats, one should go through a common threat modeling exercise. For example, one should determine whether to worry about physical threats, such as van Eck attacks[16] and shoulder surfing (both realistic physical attacks in many circumstances). Also, one must figure out whether other users on the same machine are a possible threat.

Having a concrete threat model is especially important in a randomness infrastructure, because one needs to have a complete understanding of an attacker's capabilities in order to be able to estimate the amount of entropy in data. That is, while most people think of entropy as an absolute measure of how unknowable data is, entropy is actually relative to a particular observer. For example, before the winner of an academy award is announced, the auditors who stuffed the envelope know the result. Therefore, there is no entropy in the envelope for them. Other people (such as Hollywood insiders) may have access to polling information, meaning that there may not be as much entropy as possible in the envelope if, to the observer, some possibilities are more likely than others. Of course, there may be more entropy in the envelope for the average viewer at home.

4. PRNG Design

The two major requirements for a PRNG are efficiency, as one may wish to produce a large bulk of numbers in a small amount of time, and security. Particularly, one will wish to provide a particular level of cryptographic security, where information leakage is minimized.

The security requirement is best met with a well-analyzed block cipher in CTR mode. Such a PRNG will leak a mere one bit of information after producing $2^{n/2}$ blocks of output, where n is the block size of the cipher in bits (and assuming the key length is at least as long as the block length).

One should prefer block ciphers in CTR mode to generators based on a dedicated stream cipher. CTR mode requires only that the block cipher be a pseudo-random permutation [2], which is widely believed to be a reasonable assumption. Dedicated stream ciphers, on the other hand, need to be strong ciphers and also need to resist related key attacks. For example, due to a related key attack, the naïve use of RC4 as a PRNG is fundamentally flawed [8], even disregarding biases in the cipher [9].

Cryptographic hash functions can also be a good foundation for a PRNG. Many constructs have used MD5 or SHA1 in this capacity, but the constructions are often ad hoc. When using a hash function, we would recommend HMAC in CTR mode (i.e., one MACs counters for each successive output block). Ultimately, we prefer the use of block ciphers, as they are generally better-studied constructs.

Depending on the threat model, one may wish to consider protected memory, which is difficult to ensure. See [11] for a good discussion of the issues.

4.1 Reseeding a PRNG

There are two primary reasons why one might want to reseed a pseudo-random number generator. First, the generator may not contain as much entropy as is desired, perhaps due to the system not having been seeded yet, the entropy harvester being too liberal, or due to environmental compromise. In these cases, one would like to add additional entropy to the generator state, while preserving any entropy that does exist in the current state. The other case is to provide forward secrecy. Consider an OS-based PRNG where the operating system is compromised. A generator should ensure that PRNG output requests that were fulfilled prior to the compromise cannot feasibly be recovered algorithmically from the current generator state.

The first type of reseed we call an external reseed, whereas the second is called a self-reseed.

Let us consider the case where one is using AES-128 in CTR mode as a PRNG. Reseeding the generator must involve replacing the key with a new value. Additionally, one should replace the value of

the counter with something unpredictable to the attacker, because randomizing the counter can prevent meet-in-the-middle attacks that would reduce the effective strength of the PRNG to 64 bits.

For a self-reseed one can use the generator to output 256 bits of data, then use the first 128 bits to rekey the cipher, and the second 128 bits to replace the counter value. Rekeying the block cipher with its own output is an effective one-way transformation assuming the generator has the desired entropy. This technique was first introduced in [12].

For an external reseed, one should combine new seed material with the current generator state. An external reseed can be identical to a self-reseed, except that the seed would be XORed with the mixed generator state. For example, if the seed is limited to 256 bits in length when using AES-128-CTR, one may take 256 bits of generator output, XOR in the new seed material, then rekey and replace the counter block. If the generator has not previously been seeded, one would simply pad the seed material to 256 bits, then key and set the counter block.

We recommend the seed material be taken from an entropy harvester. It should also be at least 100 bits. We recommend a 256-bit seed for the initial seeding, but subsequent seeds should be no more than 128 bits in length, as additional entropy will generally be better applied if it is saved for other applications.

If a seed is too long, it can be compressed with a cryptographic hash function. However, the entropy harvester should deal with this issue, as it should be the entity responsible for producing whitened data where the contents should be as close as random as possible.

Self-reseeds should occur after each output request so that forward secrecy can be maintained. That is, if a client requests any amount of data, the generator should provide it, and then reseed. Yarrow recommended a configurable parameter for the amount of output between self-reseeds. The single output request can be considered atomic, but we see no reason to ever accept multiple requests between reseeds, because, with a well-selected PRNG, a reseed operation is so cheap that there should never be an excuse not to use it.

For example, if using AES in CTR mode as a PRNG, reseeding would consist of incrementing and encrypting a counter and then performing key expansion on the result. As shown by timing data from Helger Lipmaa,[1] freely available AES implementations can perform the key expansion in about 200 cycles on a Pentium III, which is a trivial number. Based on Lipmaa's data, the total cost should never exceed 800 Pentium III cycles.

Determining when to perform an external reseed is more complicated. The goal of the external reseed is to put the PRNG into a secure state. Once the PRNG is likely to be in a secure state, one needs to determine whether entropy is better spent increasing assurance about the security of the PRNG state or serving clients wanting information theoretic security.

Many systems, such as Yarrow[12] and Fortuna[7], assume that all entropy in a system will be used to seed a PRNG, and try to assume that the PRNG is eventually secure.

Yarrow collects entropy in two pools, one that is used to reseed the generator frequently, in hopes that any compromise can be recovered from quickly once the threat to the system is gone, and another that does so rarely, saving up entropy over a long period of time. The idea behind the slower reseed is that the entropy estimators may be broken in some way, and it should still be possible to recover eventually, if so.

Fortuna uses 32 entropy pools. Samples from any given source are fed into the pools in a round robin fashion. A reseed occurs when the first pool is long enough. The first pool is always used to reseed the generator. The second pool is also used every other time. The third pool is additionally used every fourth time, and so on. The Fortuna solution will eventually put the system into a secure state. Fortuna is designed not to have the ability to output entropy separately. All entropy is fed into these pools, which are only used to reseed a PRNG. Most general-purpose systems will find this to be an unacceptable use of entropy.

The goal of the Fortuna design is to avoid entropy estimation. One can simply specify a length at which the first pool reseeds, and then the system will eventually make it into a secure state. There is no way of knowing when that will be without a good estimator, particularly when some sources may be tainted.

One cannot easily estimate whether the PRNG is likely to be secure based on the number of reseeds. For example, a single polluted source can feed enough entropy to cause 100 reseeds before the first good source adds any entropy to the first pool. If a polluted source causes many reseeds, then it can ensure that the first few pools always have small amounts of actual entropy when they reseed, meaning that an attacker can prevent the PRNG from reaching a secure state for a long time. Nonetheless, Fortuna will eventually get there.

Either Fortuna or Yarrow could be used when not all entropy is fed to the PRNG infrastructure. However, we still do not believe Fortuna to be a practical system, because it uses its entropy way too sparingly in order to avoid the need for entropy estimators, and does not adequately protect against short-term compromises due to the lack of entropy

[1] http://www.tcs.hut.fi/~helger/aes/rijndael.html

estimators. In Section 5, we will discuss best practices for entropy estimation.

We prefer the Yarrow approach, albeit with a different set of metrics. Yarrow's conservative reseeds happen linearly. Every time two or more entropy sources contribute more than 160 bits of entropy towards a conservative reseed, that reseed occurs.

We believe that entropy metrics involving trust of multiple sources are better placed inside the entropy harvester. Additionally, conservative reseeds should get more conservative over time.

The approach that we used in EGADS is similar to Yarrow in that we take some of the entropy sent to the PRNG and use it for fast reseeds in order to recover from compromise quickly (we discuss this problem more below). Occasionally (usually when we have done a fast reseed within a small window of time, say 60 seconds), we take entropy fed to the PRNG and feed it into a slow reseed pool, much like Yarrow.

The major difference is the metric for performing the catastrophic reseed. The approach is far closer to Fortuna, in that reseeds get more conservative on an exponential scale. The reseed pool is used as the seed to update the PRNG, but only after 2^n updates of the reseed pool, where n is the number of previous PRNG reseeds that have occurred. Using this approach, one would rely on entropy estimators in the harvesting infrastructure to prevent Fortuna-like weaknesses.

Another approach that can be used in tandem for recovering from environment compromise is to assume that the administrator will reboot as a concluding step in recovering from any attack. As a result, when reseeding after machine startup, one should stretch to find in any little shred of entropy that may be available, including timestamps surrounding portions of the boot process (even though on many systems, this information is publicly leaked). Additionally, we believe that one should always take the first 128 bits of entropy gathered by the randomness infrastructure (using an appropriate output threshold; see below) and use it to reseed the PRNG as soon as possible.

This solution leave open a window after boot where the system pseudo-random number generator is potentially vulnerable to the attacker who originally compromised the system. To avoid this problem, one could have the PRNG block until the initial reseeding. However, this seems impractical for most systems (i.e., we believe it is a design decision that would enrage many people who count on their PRNG to never block). We believe a more realistic solution is to leave this issue as an operational one. If the administrator is worried about the problem, let her boot in single user mode and ensure the generator is well seeded before putting the machine back on a network.

4.2 Dealing with fork()

A common error in application-level generators is that PRNG state may be unintentionally duplicated as the result of an operation such as `fork()`. If the developer is fastidious when using `fork()`, the following pseudo-C code solution will suffice in a non-threaded program (it has a race condition in threaded programs that one would have to ensure cannot occur):

```
if ((pid = fork())) {
 prng_output_seed();
} else {
 prng_reseed(prng_output_seed());
}
```

When forking, the parent discards the amount of data it needs for a reseed. The child will take the exact same data, and use it to reseed itself. Since the reseed is effectively a one-way transformation, the compromise of the child will not reveal the state of the parent. For the reverse to be true, the parent should also reseed itself after discarding the necessary data.

This solution works as long as the developer is diligent, which can be particularly tough in a threaded application. However, the PRNG implementer should not expect the developer to be diligent. Instead, the PRNG should detect forks and act appropriately when they occur, reseeding before generating any output.

Reseeding in such a way that the parent and child end up with different states where the output does not risk being correlated easy is non-obvious.

Previously, [11] stated that using a database-style two-phase commit was the only way to solve this problem. Here is a much simpler solution, expressed in the following pseudo-C code:

```
prng_output(numbytes) {
 my_pid = getpid();
 time  = time();
 val = concatenate(my_pid, time);
 if (saved_pid != my_pid) {
  seed = MAC(seed, ctr, val);
  saved_pid = my_pid;
 }
 increment_counter();
 /* Assume that the counter is properly   */
 /* incremented after any blk cipher op . */
 ret = ctr_keystrm(seed, ctr, numbytes);
 prng_reseed();
 return ret;
}
```

In the above code, we assume that we're using a block cipher in counter mode. If not, `ctr` can be effectively ignored. We also assume a MAC that takes three parameters, a key, a nonce and data to MAC. If there is no nonce, then concatenate the second parameter to the plaintext (or ignore it totally if not using a block cipher in counter mode).

By MACing the current PID and mixing in the time, we are effectively randomizing the seed in a way

that is unique to that child (the time is mixed in because a child could eventually fork off a second process with the same pid as an old process). Even if we fork multiple children at the same time, they will each end up with a different seed.

Once the parent finally reseeds (either by someone requesting a PRNG output, by a timed reseed or by a message from the child asking the parent to reseed), the child PRNGs cannot be compromised as a result of a compromise in the parent. To decrease the risk involved with any such window of vulnerability, the child can try to gather its own entropy and reseed itself using that entropy.

4.3 The Initial Seeding Problem

One of the biggest practical problems for operating system vendors is how to bootstrap a system-wide PRNG. Generally, several processes that run early on a machine need strong pseudo-random numbers. Once a system-wide generator has reached a secure state a single time, the system can maintain a seed file across reboots (where a seed file would stash enough data to reseed the PRNG using an algorithm such as the one discussed earlier). As long as the seed file is properly protected through operational means, and assuming no fundamental flaws with the randomness infrastructure, then this approach is sound.

However, it still leaves the question of obtaining entropy the first time a machine boots. Developers of random devices for common operating systems are yet to address this problem. Instead, they generally ignore the issue, allowing the PRNG to output without sufficient entropy.

For desktop or server-based OSes, collecting entropy during the install process addresses this problem well, since most installs are interactive. We recommend explicitly prompting the user to introduce entropy into the system, such as by shaking a mouse or pounding on a keyboard.

Getting that initial entropy can be a lot more difficult in embedded devices. Under NDA, we have heard horror stories of such devices rolling off the lines with every device producing identical random numbers, or highly correlated numbers. In this space, one generally has far less opportunity to collect entropy and is greatly concerned about manufacturing cost.

Often, devices need to be individualized in order to hold a unique secret key. In such a case, it should be reasonable to generate a secure seed on another device where entropy is readily available, and then put it into devices at manufacture time (the individualized key itself should be high entropy and can be used as a PRNG seed itself). However, sometimes individualizing a device at manufacture time is too expensive.

For some devices, it may be possible to use a similar trick when the device is first deployed. For example, if a device has external buttons, the user could be asked to push the buttons randomly until the device beeps as a part of setup.

Often, such a trick will not be feasible. In such a case, the device must necessarily try to do some entropy collection on its own. Hardware solutions exist, but increase costs.

If a pure software solution is an absolute requirement, one is stuck trying to find the best available entropy. Often, this is going to be whatever clock is available. In such a device, it may be reasonable to assume that mixing in tiny bits of entropy as it becomes available is better than collecting entropy until a threshold is reached, and then reseeding all at once (which requires more overhead). Ongoing compromise of the physical device (i.e., an iterative attack on the system) is often not in the threat model, in which case, both approaches will have the same end effect. Executing on this assumption ensures that, when a device finally does collect enough entropy to be in a secure state, it will be in a secure state at its earliest opportunity. Additionally, it helps protect the seed used at time of manufacture.

5. Entropy Harvester Design

Design of a secure PRNG is a relatively straightforward matter under the assumption that occasionally there are external seeds available, where an entropy harvester is responsible for assurance as to the amount of entropy in those seeds.

All of the real challenges lie in building a good entropy harvester. Such an infrastructure needs to gather, compress and store data from its surrounding environment that may contain entropy, estimate how much entropy it is holding, whiten the entropy to remove any lingering patterns and output high-entropy data using metrics based on its estimates. All of these things are a challenge in their own right, but it is also important to do each of these things quickly.

One important design principle that should be applied to entropy harvesting infrastructures is defense in depth. In particular, if entropy harvester outputs turn out not to be secure in the information theoretic sense, then the system should be able to fall back to computational security if at all possible. Data whitening is using cryptography to remove statistical correlations between bits, effectively spreading entropy evenly throughout data. This is commonly done with a cryptographic hash function, but can be done with a block cipher. In fact, it should be a requirement of any public random number interface, though is not necessary when keying a cryptographic PRNG). Note that the output of the hash function must be truncated to a size that corresponds with the amount of entropy believed to be in the input.

5.1 Accumulating and yielding entropy

In many systems, entropy samples may be processed frequently, such as when there is a flurry of activity on the machine. The developers of OS-level PRNGs have strived to minimize the cost of processing entropy samples. Generally, such generators are only willing to keep fixed-size internal buffers.

Yarrow and Fortuna both recommend keeping one or more cryptographic hash contexts. Most operating systems have rejected this approach because they believe it is too slow. We have done timing tests, comparing the LFSR used to collect entropy in Linux's implementation of /dev/random to MD5, SHA1 and SHA256 contexts being used to process the same samples. We also considered hash127[3], a non-cryptographic hash function we will discuss below.

We measured the average time to process a 16-byte sample, over 10,000 samples, using the fastest implementations of the algorithms we could find. Testing was done on a Pentium III. The results are shown in Table 1.

Algorithm	Speed (cycles per byte)
SHA256	116 cpb
SHA1	24.5 cpb
MD5	18.4 cpb
Linux LFSR	13.3 cpb
Hash127	4.0 cpb

Table 1: P3 speed for entropy sample processing

Note that we did not consider initialization costs for any of the above algorithms. SHA256, SHA1 and MD5 are significantly more expensive at startup than the other two algorithms (by hundreds of cycles). However, the Linux /dev/random implementation runs a cryptographic hash function in its entirety over the LFSR every time entropy is output. What OS developers are more concerned with is minimizing the overhead when entropy is being gathered, not when it is being requested.

Clearly, the SHA256 implementation (the recommended cryptographic hash function to use with Fortuna) does not look like a good choice for speed-critical environments.

While SHA1 and MD5 are slower than the current Linux solution, they do have the advantage that they are believed to be functions that are, for the most part, entropy preserving. That is, if one puts data of an arbitrary length containing 128 bits of entropy into a SHA1 context, the 160-bit output should have nearly 128 bits of entropy. If that property does not hold, then the cryptographic hash function is not doing its job. Currently, SHA1 is higher assurance than MD5, due to collisions found in the MD5 compression function. However, both have been better analyzed for their ability to preserve entropy than has the non-cryptographic LFSR used by Linux.

A cryptographic hash function is certainly not necessary for entropy accumulation. However, any good function must be mostly entropy preserving, even if the attacker controls part of the input. Another possibility not considered in previous systems is to use a good universal hash function[18], which would allow one to bound the probability of significant entropy loss to something miniscule, under the assumption that the input contains enough entropy.

The hash function we examined, hash127 consists of multiplications in the Galois Field of 2^{127}-1, which are trivial to implement. The reference implementation is incredibly fast due to several performance tricks, the most effective involving precomputation. The overhead of the precomputation should be acceptable for any desktop OS (particularly since the amount of precomputation performed can be reduced arbitrarily, if necessary, trading off a bit of speed). We estimate that a well-optimized version that uses no precomputation would run, at worst, 100% slower than the hash127 reference implementation, which is still significantly faster than the Linux solution, where there is no concrete analysis on entropy loss during compression. See [14] for a more detailed discussion of software performance for this class of hash functions.

There are some considerations in using a universal hash function in this role. First, hash127 and its ilk are keyed hash functions, meaning that a key (traditionally secret) is used as part of the computation. The more entropy in the key, the better (though one is protected if the inputs contain the estimated amount of entropy). If one has a cryptographic PRNG that was seeded with some amount of entropy, then the hash function should be keyed with output from that PRNG (and periodically rekeyin as the PRNG reseeds is a fine idea). Otherwise, the function should be keyed with any available entropy (e.g., the system clock). The output of the hash function should be used to key a cryptographic PRNG that will be used in keying any future hash contexts.

Second, as is the case with the Linux LFSR, the output of a universal hash function should never be exposed directly to an untrusted user. To prevent any sort of correlation attack, the output must be passed through a pseudo-random (whitening) function. Post-processing hash127 outputs with AES is quicker than using either SHA1 or MD5 due to the fixed-cost overhead of those two algorithms.

If using AES in counter mode for a PRNG, one could reuse the PRNG's keyed AES context. However, to prevent theoretical reductions in strength, one should ensure that inputs from the hash function and counter blocks cannot collide.

When using hash127, this is easily done. That hash function produces a 128-bit output, but the 97^{th} bit is always set to 1 (hash127's output can never hold more than 127 bits of entropy). One simply must fix the

corresponding bit of the PRNG counter block to 0. This must even be done after replacing the counter block with a random value (such as when reseeding the PRNG).

Note that you can also use cryptographic stream ciphers to postprocess universal hash functions, which is useful when using such a construct as the system PRNG. In such a situation, the postprocessing is done by taking a value of sufficient size from the generator, and then using it in a way dictated by the nature of the hash function. For example, when using hash127, the value would be added to the hash result modulo 2^{127}-1.

After applying cryptographic post-processing to any sort of entropy "pool" (accumulator), one must truncate the result to the number of bits of entropy that was believed to be in the pool (securely discarding the truncated portion). We recommend slicing off an additional bit to account for the loss that one should expect from the cryptographic post-processing.

For example, one might use hash127 to collect an estimated 121 bits of entropy at a time (where the estimate is believed to be sufficiently conservative against any threat in the threat model), AES-encrypt the result, then ignore one byte of the AES output.

5.2 Entropy estimation

Perhaps the biggest shortcoming in most practical random number generation systems is the lack of quality entropy estimation. For example, EGD, a popular user-space daemon for harvesting entropy estimates .1 bits of entropy per byte of output any time it calls the `ps` command. That means a 6,000 byte `ps` output will be estimated to have 600 bits of entropy. We have found through empirical testing that this is a gross overestimate. Even if the threat model is that an attacker got to see the `ps` output a single time, and then must guess any changes based on external behavior, we've seen many situations where the an output of that size would contain no more than two bits of entropy. Particularly, machines without interactive users often have a fairly fixed set of processes, and the information about those processes that the command displays tends to change slowly. The only thing that changes frequently is CPU time elapsed, which tends to grow at an easily predictable rate.

Of course, from the point of view of a remote attacker who must guess the state of the entire process table, there may be a bit more entropy. It's difficult to say how much, because an attacker might be able to make incredibly educated guesses about the state of a system, particularly considering that the nmap tool can often give the system uptime of a remote host (depending on the operating system and the firewall configuration of the host being targeted). In short, if we were only concerned about collecting data that contains entropy with respect to remote attackers, we might be inclined to assume some amount of entropy per process (probably a fraction of a bit). Certainly, we would not expect to gain much additional entropy as time goes on (and, any additional entropy is likely to come primarily from new processed being added and possibly old processes going away—entropy is not as likely to appear in the output simply from old processes continuing to exist).

When considering threat models where local users are a threat, one must consider the risk of an attacker measuring the same data as the process that does entropy gathering. Even if the entropy gathering is done in-kernel, the end user can often make measurements that can help reveal data that is generally expected to be private to the kernel. For example, an OS that measures timestamps associated with keystrokes may be able to hide key press information between users, but the same user should be able to add her own keystrokes, which the operating system will generally assume is entropy. Unfortunately, the attacker can generally collect timestamps in user space associated with those keystrokes that are highly correlated with the timestamps of the kernel.

One general principle for entropy gathering that is widely practices is mixing a timestamp into the entropy accumulator when adding a sample. This seems like it would be an effective measure, especially considering the high-resolution timers available on modern architectures, which tend to be tied to the clock speed of the processor. However, the end user generally does not see anywhere near the maximum resolution from a timer, because most events will be regulated by the bus clock, which is generally much slower than the processor clock. Making matters worse, peripheral clocks tend to be even slower still. For example, keyboards and mice generally use a 1KHz clock, a far cry from the 3 GHz clock available on some x86 processors. As a result, when estimating entropy from a source, we recommend assuming that the associated timestamp has no better granularity than that of the slowest clock to which the entropy sample may be tied.

A related issue is a "back-to-back" attack where, depending on the details of entropy events, an attacker may be able to force events to happen at particular moments. For example, back-to-back short network packets can keep a machine from processing keyboard or mouse interrupts until the time when it is done servicing a packet, which a remote attacker can measure by observing the change in response in the packets he sends. In this particular situation, one can thwart the problem by assuming there is no entropy when the delta between two events is close to the interrupt latency time. This works because both network packets and keystrokes cause interrupts.[2]

[2] Some OSes can mitigate this problem if supported by the network card.

One significant problem is a lack of methodology for deriving reasonable estimates. The most practical methodology to date has been Yarrow's approach, which consists of applying three estimators to each input sample, and choosing the lowest estimate. The first is a programmer driven estimate. The methodology with which the programmer should derive such estimates is unspecified. The second estimator is a statistical estimator that "is geared towards detecting abnormal situations in which the samples have very low entropy". Again, the methodology for deriving such estimators is unspecified, and this estimator is not clearly differentiated from the first. The third estimator is multiplying the input length of a sample by a fixed system-wide value. The recommended value is 0.5. No justification for this value is given.

Information theory does provide ways to measure entropy, but they are not practical, because one can only model how much entropy is in data if one has a complete understanding of how the data is produced and all possible channels an attacker may use to measure information about the data. Considering that there are a broad range of possible threat models, and considering that machines behave deterministically yet are still incredibly complex in practice, one should expect data to tend to be predictable (the only times where significant new entropy can really added be to a system are when the machine receives external input), yet it is incredibly difficult to figure out just how predictable.

The most useful way to think about the maximum entropy that data can have is by looking at how much the ordered samples could be compressed (this value is clearly an upper bound on the amount of entropy in a piece of data).

[11] discusses how to use a general-purpose compression function for getting an upper bound on the amount of entropy in the data. Essentially, one compresses the first sample padded out to the internal block length of the compression function. Then, one compresses subsequent samples, and measures the upper bound on the entropy in the second sample by looking at the compressed size of that sample. The first sample is never credited for any entropy (at least, not through compression-based techniques).

The problem with this approach is that "general purpose" compression functions are not optimized for compressing specific types of data. Hand-tailored compression functions can generally do better.

We recommend one builds hand-tailored compression functions, and then attempting further compression on the output of the hand-tailored output by using a conventional compression algorithm. If the compression output does not usually grow, then the hand-tailored algorithm is clearly poor, but the combined algorithm is better.

When combining compression algorithms, the upper bound on entropy is the minimum compressed size of the sample at any point in the process. In practice, one should assume that smarter compression could do a better job, even with a hand-tailored compression function.

Our recommendation is that one should try to find as compact a representation as possible for expressing the differences between any two successive samples. This type of compression function does not take into account long-term trends on data. However, if one finds an optimal function of this type, it is rare that one will see better than a 50% improvement in compression rates by using long-term information. Also, when longer-term patterns would produce massive improvements in compression, it is likely to be obvious to the person writing the compression function. When building such a compression function, one should conservatively assume that data that changes predictably (such as CPU time in the output of `ps`) is constant. Additionally, one should ignore any data that an attacker might be able to measure (assuming a particular threat model).

With any compression function that one uses that seems to give good results, we recommend conservatively dividing the entropy estimates derived from such a function by four. This helps hedge against the existence of better compression algorithms as well as the presence of unanticipated channels of attack.

The next question that arises is whether entropy measurement should be done "on the fly", or whether a static estimator should be used.

While dynamic estimators certainly have more overhead, there is a big danger in a static estimator, in that one may end up with entropy samples that have less entropy than the estimate. For example, no matter what estimate one were to derive for keyboard entropy, it would not be low enough for the case when a key is being held down (assuming a highly conservative threat model).

FIPS-style statistical tests (see Section 5.4) provide a solution, but such tests are not satisfying because they are easily fooled. One option is to derive a set of conditions where the estimate used is believed to be too liberal, and then detect those conditions, estimating no entropy for them. For example, one might fail to credit entropy any time the same key is pressed twice in a row, or any time an interrupt-based event occurs back-to-back with another such event.

Another thing we recommend is to set a maximum entropy estimate per sample for any given source. We recommend examining a source in a variety of operational environments and calculating a median compression rate, then basing a maximum off that value (for example, divide by four).

These guidelines should lead to per-source entropy estimators that are reasonable if a high-quality

compression function is found and if the channels over which an attacker might get data are well understood. We would recommend this approach over the more ambiguous guidelines required by Yarrow.

A per-source, per-sample maximum that is carefully calculated seems more appropriate than a Yarrow-like global factor based on the length of the input. Otherwise, the spirit of the Yarrow estimators is present in the above recommendations.

5.3 Metrics for yielding entropy

Even if all entropy estimators in an entropy harvesting system are believed to be adequately pessimistic under a particular threat model, a good conservative approach would assume that at least one entropy source has an unanticipated side channel. This was first done in Yarrow, where a catastrophic reseed occurred when the appropriate entropy pool had built up at least 160 bits of entropy from `k` different sources, where `k` was expected to be two in practice.

Such an approach was only used for catastrophic reseeds. We feel that a similar approach should be used for any entropy harvester output, even if not being used for catastrophic reseed, since the goal of an entropy harvester is to provide information theoretic security.

However, the Yarrow metric is somewhat wasteful in this capacity, in that it eats at least 320 bits of entropy to produce a single 160-bit output, and will often eat quite a bit more, particularly when there is a single, fast source (the metric is more appropriate for the context in which it is used).

We believe more liberal metrics are appropriate, particularly because previous designs favored by implementers, such as the Linux design, are widely believed to do a good enough job in practice. We have heard no reports of someone breaking the Linux system by polluting an entropy source (either `/dev/random` or `/dev/urandom`). Considering that such a design seems to be good enough, if entropy outputs are a requirement of a system, then a Fortuna-style design seems like overkill (particularly when a fast, polluted source can stall a secure PRNG seed for quite a long time).

There are plenty of more liberal metrics that can be more effective. One example would be, for an n-bit accumulator, to mark as ready for output when it contains an estimated `n` bits of entropy, disregarding the contributions of the fastest `k` entropy sources (where `k` will generally be 1 in most practical systems).

In such a scenario, any single source would stop adding entropy to a particular accumulator after contributing `n` bits. The source could then move on to another accumulator (when all accumulators are full, one should be emptied into the PRNG, as discussed above).

A more liberal metric still is possible if one is willing to fall back on the computational security provided by the whitening process. For example, let us assume that `n`=128 bits. A metric could be that, either the accumulator has 128 bits of entropy ignoring the most prolific source, or that at least two sources have contributed 64 bits of entropy. In this way, the system should approximate information theoretic security, unless one of the sources is tainted, in which case the 64 bits of entropy should still provide reasonable protection in practice.

Other metrics can be designed that are tailored to the needs of a particular environment. One might assign trust rankings to each source, and output when the entropy buffer achieves a particular level of trust. That way, a fast, trusted source (such as a hardware pseudo-random number generator) can still be used to generate entropy outputs quickly. Or, one could simply grant an exception to trusted sources.

Another issue in entropy estimation that has never been addressed previously is the possibility of partially tainted sources on multi-user machines. For example, on a multi-user system, measuring entropy for keyboard interrupts can be partially tainted if there is a single malicious user. If multiple sources can be tainted in this way at the same time, the above metrics can easily fail.

One could solve this problem by keeping track of each entropy source on a per-user basis, treating each as a logical source. However, that technique has the problem in practice that it requires dynamic allocation of entropy estimation information, which may conceivably pose a problem.

5.4 Other Considerations

Much like a PRNG, an entropy infrastructure should be able to provide forward secrecy in the case of environmental compromise. Such a goal is easily achieved by zeroing out entropy accumulators securely after they are used for output. This can also help ensure that outputs derived from the same accumulator are sufficiently independent that information-theoretic security levels can still be approximated, which is particularly important when using an accumulator without theoretically proven bounds for entropy loss (such as the construct used with Linux' `/dev/random`).

Besides the Linux solution not having provable information theoretic security, its methodology of hashing its entropy pool to get (hopefully) entropic output, stirring the pool and then reducing the entropy estimate by the number of bytes output is a waste from the perspective of cryptographic security, since there are constructs where information leakage is both miniscule and well bounded.

Recovering from an environmental attack is somewhat more problematic. If computational security is an acceptable fallback for outputs, then the whitening process can involve PRNG output, so that once the PRNG returns to a computationally secure state, entropy harvester outputs will do so as well. For example, one might take a 128-bit key from the PRNG and then encrypt a 128-bit accumulator using that key before outputting.

If computational security is not a desirable fallback, then we recommend assuming that a machine will be rebooted after recovering from a compromise, and then being sure not to save the state of entropy accumulators across a reboot (instead, we would mix the accumulators into the PRNG state before shutting down).

FIPS 140-1 tests[21] are worthwhile to perform in an entropy harvester, despite the fact that there may be low-entropy output streams they will not fail. Such tests should be applied to the output of each source, in the hopes of detecting when the source has a catastrophic failure. If they do not pose significant overhead, we recommend one run them continuously on compressed entropy streams before they are placed in an accumulation buffer (at that point, the FIPS tests are unlikely to be overly useful). Otherwise, one should run them whenever cycles to do so are available.

As with PRNGs, the memory used to hold all the state of an entropy harvester should be protected using the best available means.

Timing and power-based side channels can be extremely difficult to avoid, considering the wide disparity in entropy sources (in the rest of the infrastructure, it should be fairly simple to avoid conditional branches and loops of indefinite size). Such problems argue for using a general-purpose compression function that may be less accurate, but also less susceptible to side channel attacks. This area is in need of additional research.

For user-space entropy accumulators, the PRNG `fork()` issue also applies to entropy accumulators. Here, the state problem is easy to solve, as the child should simply zero out all of its entropy buffers. However, one should note that there is significant risk of parent and child measuring redundant entropy, in which case estimates may be significantly off. This argues for system-wide entropy accumulators.

In the situation where a user-space collector is believed to be higher assurance than an existing kernel-based generator, then the user-space generator should be sure to estimate under the assumption that the kernel generator is leaking significant amounts of information as to the date that the user-space collector is gathering.

6. Related Work

Traditionally, design of randomness architectures focused on the PRNG, and not the collection and management of entropy. Many systems exist where the PRNG itself is cryptographically strong (or is believed to be), where the responsibility for providing a secure seed is left to the client of the PRNG, including the ANSI X9.17 PRNG[20] and the Blum-Blum-Shub generator[4], which is based on public key cryptography, but is no more secure and is far slower in practice than PRNGs based on a block cipher in CTR mode (both constructs are provably secure to good bounds with reasonable assumptions). Even [9], which recognizes the fundamental difference between information theoretic security and computational security, assumes that clients will only ever want computational security. Even so, it misses PRNG requirements we believe to be important, such as forward secrecy. Yarrow[12] and Fortuna[7] are the only published PRNGs to recognize forward secrecy as a requirement.

[6] famously discusses ways to harvest entropy, and explicitly talks about stretching random numbers with a pseudo-random number generator when "true" random numbers cannot be obtained quickly enough. [11] discusses specific considerations in polling for entropy and discusses using compression for entropy estimation, but totally ignores conservative Yarrow-style metrics. [17] analyzes common sources for entropy in software, and provides lower-bound metrics that are probably suitably conservative for general-purpose operating systems.

The original Linux infrastructure for `/dev/random` and `/dev/urandom` was the first interface to separate the notion of high-assurance randomness (i.e., data obtained from `/dev/random`) and random data that may not be secure (but generally is in practice; the `/dev/random` interface). The original system worked by pooling entropy in a LFSR, and estimating the number of bits in that LFSR. When output was requested, the pool was "stirred" (the LFSR is clocked) and then hashed with MD5, which was used for up to 128 bits of output, then 128 was subtracted from the entropy count of the pool. If output was requested over the `/dev/random` interface, yet not enough entropy was estimated to be in the pool, then output blocked until that condition was true.

This general system persists to this day (though it has evolved a bit, even incorporating a Yarrow-inspired slow pool), and has been widely copied, with minor modifications. Unfortunately, while it might seem to give information theoretic security due to limiting output to the amount of entropy believed to be in the pool, it does not do so, because subsequent outputs are

not generated using independent entropy. The same piece of data is rotated and rehashed each time, until the entropy estimate reaches zero. As a result, if an attacker could invert the hash function, then she could calculate the next output with high probability by rotating the pool and rehashing. The only time this would fail is if new entropy gets added to the pool in the meantime.

One consequence of this design is that the `/dev/random` device is not suitable for producing keys larger than the output size of the hash function used internally, as the only case where such a key could possibly have more than 160 bits of entropy would be when enough new entropy got added to the pool in-between blocks of output.

Because `/dev/random` on Linux does not provide information theoretic security, we see no reason to use it compared to `/dev/urandom`, as long as one can ensure that the entropy pool has ever received a minimum amount of entropy. The entropy loss through `/dev/urandom` is miniscule, and therefore there is no good reason to ever block for more entropy if information-theoretic levels of security cannot be ensured.

In the original Linux random number infrastructure, malicious applications could starve `/dev/random` by continually reading from `/dev/urandom`. Of course, malicious applications could also try to read continually from `/dev/random`. A partial solution to that problem is to provide data on the `/dev/random` interface in a round-robin manner in fixed-size increments. This way, legitimate users will eventually get served.

There are other problems with this infrastructure. For example `/dev/random` output bits are not necessarily likely to be secure in the information-theoretic sense, partially due to the lack of proofs attached to the LFSR and partially due to the fact that the LFSR is not zeroed out after each output. Also, the way /dev/random was designed made it difficult to analyze. The LFSR appears to be a reasonable construct for accumulating entropy, but it may not be.

Yarrow is criticized for being too complex and under-specified, particularly with regard to its entropy handling. Several implementations intended for deployment have been so slow to provide an initial seed that they were shelved.

Fortuna is even more reticent to use its entropy effectively, and completely foregoes entropy estimation, which we believe is theoretically appealing, but a drawback in practice. Neither Fortuna nor Yarrow allow for an interface to data with information-theoretic security levels.

The ideas presented in this paper are implemented in the EGADS software package. This work extends a previous, unpublished algorithm called Tiny, developed in conjunction with John Kelsey.

7. References

[1] B. Arkin, F. Hill et al. "How we learned to cheat in online poker", developer.com, Sep. 1999.

[2] M. Bellare, A. Desai, E. Jokipii and P. Rogaway, "A concrete security treatment of symmetric encryption: analysis of the DES modes of operation", In Proc. 38th Annual Symposium on Foundations of Computer Science, 1997.

[3] D. Bernstein, "Floating-point arithmetic and message authentication", at http://cr.yp.to/papers.html#hash127, 2000.

[4] L. Blum, M. Blum, and M. Shub. "A simple unpredictable pseudo-random number generator". In SIAM Journal on Computing , 15, May 1986.

[5] A. Desai, A. Hevia and Y. Yin, "A practice-oriented treatment of pseudorandom number generators", In Proc. Eurocrypt 2002, Springer-Verlag, 2002.

[6] D. Eastlake, S. Crocker and J. Schiller, "Randomness recommendations for security", RFC 1750, Internet Engineering Task Force, Dec. 1994.

[7] N. Ferguson and B. Schneier, Practical Cryptography. Indianapolis: John Wiley & Sons, 2003.

[8] S. Fluhrer, I. Mantin and A. Shamir, "Weaknesses in the key scheduling algorithm of RC4", in Proc. Eighth Annual Workshop on Selected Areas in Cryptography, Aug. 2001.

[9] S. Fluhrer and D. McGrew, "Statistical analysis of the alleged RC4 stream cipher", in Proc. Fast Software Encryption Seventh International Workshop, Springer-Verlag, Mar. 2000.

[10] I. Goldberg and D. Wagner, "Randomness and the Netscape browser", Dr. Dobbs' Journal, Jan. 1996.

[11] P. Gutmann, "The design and verification of a cryptographic security architecture," Ph.D. dissertation, Dept. Comp. Sci., Univ. of Auckland, Aug. 2000.

[12] J. Kelsey, B. Schneier and N. Ferguson, "Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator", in Proc. Sixth Annual Workshop on Selected Areas in Cryptography, Springer Verlag, Aug. 1999.

[13] J. Kelsey, B. Schneier, et al., "Cryptanalytic attacks on pseudorandom number generators", in Proc. Fast Software Encryption, Fifth International Workshop, Mar. 1998.

[14] T. Kohno, J. Viega and D. Whiting, "The CWC Authenticated Encryption (Associated Data) Mode", ePrint Archives, 2003.

[15] B. Moeller, "OpenSSL security advisory: PRNG weakness in version up to 0.9.6a", posting to bugtraq, Jul. 2001, message-ID 20010710130317.A1949@openssl.org.

[16] W. van Eck, "Electromagnetic radiation from video display units: an eavesdropping risk?" in Computers & Security, 1985 Vol. 4.

[17] J. Viega and M. Messier, The Secure Programming Cookbook for C and C++, Sebestpool: O'Reilly and Associates, 2003.

[18] M. Wegmen and L. Carter, "New hash functions and their use in authentication and set equality", Journal of Computer and System Sciences, 22, 1981.

[19] J. Ziv and A. Lempel, "A universal algorithm for sequential data-compression", IEEE Transactions on Information Theory, 23(3), May 1977.

[20] "American National Standard for financial institution key management (wholesale)", American Bankers Association, 1985.

[21] "Security requirements for cryptographic modules", NIST, FIPS pub. 140-1, Jan. 1994.

[22] "Security requirements for cryptographic modules", NIST, FIPS pub. 140-2, Dec. 2002.

6. Pseudorandom Numbers

Jeffrey C. Lagarias
AT&T Bell Laboratories

ABSTRACT

This chapter surveys the problem of generating pseudorandom numbers. It lists many of the known constructions of pseudorandom bits. It outlines the subject of computational information theory. In this theory the fundamental object is a secure pseudorandom bit generator. Such generators are not theoretically proved to exist, although functions are known that appear to possess the required properties. In any case, pseudorandom number generators are known that work reasonably well in practice.

6.1 INTRODUCTION

A basic ingredient needed in any algorithm using randomization is a source of "random" bits. In practice, in probabilistic algorithms or Monte Carlo simulations, one uses instead "random-looking" bits. A **pseudorandom bit generator** is a deterministic method to produce from a small set of "random" bits called the **seed** a larger set of random-looking bits called pseudorandom bits.

There are several reasons for using pseudorandom bits. First, truly random bits are hard to come by. Physical sources of supposedly random bits, which rely on chaotic, dissipative processes such as varactor diodes, typically produce correlated bits rather than independent sequences of bits. Such sources also produce bits rather slowly (see Schuster, 1988). Hence

Probability and Algorithms
By Panel on Probability and Algorithms, National Research Council (U.S.). Panel on Probability and Algorithms
Published by National Academies Press, 1992
ISBN 0309047765, 9780309047760
178 pages

2.7 Generating Pseudorandom Numbers

Problem

You want to generate a random number.

Solution

The `Math.random( )` method returns a pseudorandom number between 0 and 1. To calculate a pseudorandom integer value within a range starting with zero, use the formula:

```
var result = Math.floor(Math.random( ) * (n + 1));
```

where `n` is the highest acceptable integer of the range. To calculate a pseudorandom integer number within a range starting at a number other than zero, use the formula:

```
var result = Math.floor(Math.random( ) * (n - m + 1)) + m;
```

where `m` is the lowest acceptable integer of the range, and `n` is the highest acceptable integer of the range.

Discussion

The previous examples focus on random integers, such as the kind you might use for values of a game cube (a die with numbers from 1 through 6). But you can remove the `Math.floor( )` call to let the rest of the expression create random numbers with decimal fractions if you need them.

JavaScript's random number generator does not provide a mechanism for adjusting the seed to ensure more genuine randomness. Thus, at best you can treat it as a pseudorandom number generator.

See Also

"The Math Object" in the introduction to this chapter.

2.8 Calculating Trigonometric Functions

Problem

You want to invoke a variety of trigonometric functions, perhaps for calculating animation paths of a positioned element.

Solution

JavaScript's `Math` object comes with a typical complement of functions for trigonometric calculations. Each one requires one or two arguments and returns a result in

JavaScript & DHTML Cookbook
By Danny Goodman
Published by O'Reilly, 2007
ISBN 0596514085, 9780596514082
582 pages

5.41 Note *(efficiency of the Blum-Blum-Shub PRBG)* Generating each pseudorandom bit z_i requires one modular squaring. The efficiency of the generator can be improved by extracting the j least significant bits of x_i in step 3.2, where $j = c \lg \lg n$ and c is a constant. Provided that n is sufficiently large, this modified generator is also cryptographically secure. For a modulus n of a fixed bitlength (eg. 1024 bits), an explicit range of values of c for which the resulting generator is cryptographically secure (cf. Remark 5.9) under the intractability assumption of the integer factorization problem has not been determined.

5.6 Notes and further references

§5.1

Chapter 3 of Knuth [692] is the definitive reference for the classic (non-cryptographic) generation of pseudorandom numbers. Knuth [692, pp.142-166] contains an extensive discussion of what it means for a sequence to be random. Lagarias [724] gives a survey of theoretical results on pseudorandom number generators. Luby [734] provides a comprehensive and rigorous overview of pseudorandom generators.

For a study of linear congruential generators (Example 5.4), see Knuth [692, pp.9-25]. Plumstead/Boyar [979, 980] showed how to predict the output of a linear congruential generator given only a few elements of the output sequence, and when the parameters a, b, and m of the generator are unknown. Boyar [180] extended her method and showed that linear *multivariate* congruential generators (having recurrence equation $x_n = a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_t x_{n-t} + b \bmod m$), and *quadratic* congruential generators (having recurrence equation $x_n = a x_{n-1}^2 + b x_{n-1} + c \bmod m$) are cryptographically insecure. Finally, Krawczyk [713] generalized these results and showed how the output of any *multivariate polynomial* congruential generator can be efficiently predicted. A *truncated* linear congruential generator is one where a fraction of the least significant bits of the x_i are discarded. Frieze et al. [427] showed that these generators can be efficiently predicted if the generator parameters a, b, and m are known. Stern [1173] extended this method to the case where only m is known. Boyar [179] presented an efficient algorithm for predicting linear congruential generators when $O(\log \log m)$ bits are discarded, and when the parameters a, b, and m are unknown. No efficient prediction algorithms are known for truncated multivariate polynomial congruential generators. For a summary of cryptanalytic attacks on congruential generators, see Brickell and Odlyzko [209, pp.523-526].

For a formal definition of a statistical test (Definition 5.5), see Yao [1258]. Fact 5.7 on the universality of the next-bit test is due to Yao [1258]. For a proof of Yao's result, see Kranakis [710] and §12.2 of Stinson [1178]. A proof of a generalization of Yao's result is given by Goldreich, Goldwasser, and Micali [468]. The notion of a cryptographically secure pseudorandom bit generator (Definition 5.8) was introduced by Blum and Micali [166]. Blum and Micali also gave a formal description of the next-bit test (Definition 5.6), and presented the first cryptographically secure pseudorandom bit generator whose security is based on the discrete logarithm problem (see page 189). Universal tests were presented by Schrift and Shamir [1103] for verifying the assumed properties of a pseudorandom generator whose output sequences are not necessarily uniformly distributed.

The first provably secure pseudorandom number generator was proposed by Shamir [1112]. Shamir proved that predicting the next number of an output sequence of this generator is equivalent to inverting the RSA function. However, even though the numbers as a whole may be unpredictable, certain parts of the number (for example, its least significant bit) may

Handbook of Applied Cryptography
By Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone
Published by CRC Press, 1997
ISBN 0849385237, 9780849385230
780 pages

Random Numbers

Computer programs are deterministic: they execute a fixed sequence of instructions. Given a particular input, they produce the same output every time. How, then, can they behave randomly? They can't, really. When your program needs a random number, the best it can do is request a *pseudorandom* number.

Pseudorandom numbers are produced by simple chunks of software called *pseudorandom number generators*. Perl's built-in generator is called rand(), and it either uses the internal rand function of the operating system or a rand function that comes with the Perl source. When given an argument, rand() produces a floating-point number between 0 and that argument. For example, `1 + int( rand(6) )` "rolls a die", that code returns an integer between 1 and 6. rand() is usually a *linear congruential generator*.* These functions take an integer, multiply it by one constant, add another constant, and ignore any integer overflow. The result is the random number (typically divided by the maximum integer size, yielding a floating-point number between 0 and 1), and is used as the input for the next invocation of the random number generator. The constants have to be chosen carefully to mesh with the word size of the computer and to ensure that the random numbers don't repeat for a very long time (say, 2^{32} calls to rand()).

Pseudorandom number generators allow you to predict the second random number from the first, the third random number from the second, and so on. Once you know the first integer, you can predict every single number your program will ever generate. So where does the first integer come from?

Don't Forget to Seed Your Generator

The input to the first invocation of a pseudorandom number generator is called the *seed*. In Perl, you can choose the seed with the built-in srand() function. In Perl Version 5.004 and later, if you haven't called srand() before the first call to rand(), Perl automatically calls srand() for you using a value that is very hard to predict:

```
srand;          # Before Perl 5.004, this did srand(time);
                #     Starting with 5.004, it uses a mix of values
                #     that should be very hard to predict.
print rand;     # A random floating-point number between 0 and 1
```

You might wish to seed with a remembered value if you want to be able to duplicate the sequence of values returned by rand from another run, but normally the

* For a description of linear congruential generators and other ways of generating random numbers in Perl, see Jon Orwant's "Randomness" in *The Perl Journal* Issue #4 and Conrad Lund's "Random Number Generators and XS" in *The Perl Journal* Issue #6. *Numerical Recipes in C* has a more in-depth treatment of the strengths and weaknesses of pseudorandom number generators.

Mastering Algorithms with Perl: Jon Orwant, Jarkko Hietaniemi, John Macdonald
By Jon Orwant, Jarkko Hietaniemi, John Macdonald
Published by O'Reilly, 1999
ISBN 1565923987, 9781565923980
684 pages

Exercises 13.2

1. Show that, in the Goldwasser–Micali scheme, it takes $O(k^2)$ time to encrypt each bit and that it takes $O(k^3)$ time to decrypt each bit.
2. In the GM-scheme, a receiver A has public key $N = 35$, $y = 3$. How would she decode the cipher text (29, 13)?

13.3 Cryptographically secure pseudorandom numbers

The problem of finding 'unpredictable' pseudorandom numbers is intimately related to randomized encryption. Given a method of generating such sequences, it can be used as the source of approximate one-time pads that are safe in the way that the linear shift-register sequences described earlier are not.

A secure probabilistic encryption scheme, based on the ideas of Goldwasser and Micali but much more efficient to implement, has been announced recently by Blum and Goldwasser (1985). The basic idea of the scheme is to encrypt the message by taking its module 2 sum with the output of an unpredictable pseudorandom sequence generator.

Suppose A and B wish to communicate secretly (and therefore share) a common random-number generator and a common secret seed s. In order to send a block of plaintext $M_1, M_2, \ldots$ to B, sender A uses s to generate a sequence of pseudorandom numbers $X_1, X_2, \ldots$, and forms the cryptogram $C = C_1 C_2 \ldots$, where

$$C_i = M_i + X_i \pmod 2.$$

A danger to the system occurs when an enemy has some side information enabling him to find out some of the X_i. Based on these values, he may be able to generate the rest of the sequence and thus break the cryptogram.

Example One of the most popular and fast methods of obtaining pseudorandom sequences is to use the method of linear congruential generators. These consist of a *modulus* N, a *multiplier* a relatively prime to N, and an *increment* c. Starting from a random *seed* X_1, the sequence $(X_n : 1 \le n < \infty)$ is given by

$$X_{n+1} = aX_n + c \pmod N. \tag{1}$$

Thus the X_i are all integers between 0 and $N-1$. The sequences produced have been shown to satisfy many and various statistical tests of randomness for proper choices of the parameters a, c, and N (for details see Knuth, 1969). Because of this and their ease of generation, this *linear congruential* method is widely used.

Codes and Cryptography
By Dominic Welsh
Published by Oxford University Press, 1988
ISBN 0198532873, 9780198532873
257 pages

9 Quasi-Monte Carlo Methods

We next discuss *quasi-Monte Carlo* methods for integration and simulation. In contrast to Monte Carlo ideas, they rely on ideas from number theory and Fourier analysis, and they are often far more powerful than Monte Carlo methods. In this chapter we present the basic quasi-Monte Carlo ideas and methods.[1] Before we get to the details, we first make some general points.

To understand the basic idea behind quasi-Monte Carlo methods, it is useful to realize what Monte Carlo methods really are in practice. As we pointed out in chapter 8, pseudorandom sequences are generally used in practice instead of truly "random" (whatever that means) sequences. Despite their similarities, pseudorandom sequences are deterministic, not random, sequences. Mathematicians have long known this; according to John von Neumann, anyone who believes otherwise "is living in a state of sin." Since they are not probabilistically independent, neither the law of large numbers nor the central limit theorem[2] apply to pseudorandom sequences. The fact that neither linear statistical techniques nor the human eye can distinguish between random numbers and pseudorandom number sequences is of no logical relevance and has no impact on mathematical rates of convergence.

These issues are often confusingly discussed in the Monte Carlo literature. For example, Kloek and van Dijk (1978) contrast the Monte Carlo convergence rate of $N^{-1/2}$ with Bahravov's theorem showing that the convergence rate for deterministic schemes for d-dimensional C^1 integrable functions is $N^{-1/d}$ at best. However, they then go on to use pseudorandom number generators in their Monte Carlo experiments. This is typical of the Monte Carlo literature where probability theorems are often used to analyze the properties of algorithms that use *deterministic* sequences. Strictly speaking, procedures that use pseudorandom numbers are not Monte Carlo schemes; when it is desirable to make the distinction, we will refer to Monte Carlo schemes using pseudorandom sequences as *pseudo-Monte Carlo* (pMC) methods.

Even though experimental evidence of pseudo-Monte Carlo methods shows that they work as predicted by the Monte Carlo theory, this does not validate the invalid use of probabilistic arguments. Pseudorandom methods are instead validated by a substantial literature (Niederreiter 1992 and Traub and Wozniakowski 1992 are recent good examples) which explicitly recognizes the determinism of pseudorandom sequences. These analyses rely on the properties of the deterministic pseudorandom

1. In this chapter we frequently use Fourier expansions wherein the letter i will denote the square root of -1. To avoid confusion, we do not use it as an index of summation.

2. I here refer to the law of large numbers as stated in Chung (1974, theorem 5.2.2), and the central limit theorem as stated in Chung (1974, theorem 6.4.4). These theorems rely on independence properties of random variables, not just on zero correlation. There may exist similar theorems that rely solely on the serial correlation and runs properties of pseudorandom sequences, but I don't know of any.

Numerical Methods in Economics
By Kenneth L. Judd
Published by MIT Press, 1998
ISBN 0262100711, 9780262100717
633 pages